Dynamic Load Balancing with Pair Potentials

Jean-Charles Papin, Christophe Denoual, Laurent Colombet, Raymond Namyst

Dynamic Load Balancing with Pair Potentials

Jean-Charles PAPIN[1,2], Christophe DENOUAL[2], Laurent COLOMBET[2], Raymond NAMYST[3]

[1] CMLA, ENS-Cachan, 61 avenue du Président Wilson 94235 Cachan, FRANCE
[2] CEA, DAM, DIF, F-91297 Arpajon, FRANCE
[3] Université de Bordeaux, 351 cours de la Libération, 33400 Talence, FRANCE

Abstract. We present a new load balancing algorithm inspired by Molecular Dynamics Simulations. Our main motivation is to anticipate the rising costs of tasks-scheduling caused by the growth of the number of available cores on chips. This algorithm is based on a virtual decomposition of workload in Voronoï cells centered around computing units. The method used in this paper allows cores to virtually move in order to change their computing load. Cores displacements are result of forces computation (with pair potential): attractive or repulsive forces between cores are balanced by the cores computing load (total cost of Voronoï cell). Over-charged cores are more attractive than under-charged cores (which are then more repulsive). In this paper, we demonstrate the relevance of our approach by experimenting our algorithm with a high number of automatically-generated test cases, ranging from almost stable to quickly-evolving scenarii. In all cases, our algorithm is able to quickly converge to a distribution which maintains good locality properties.

Keywords: Simulation, dynamic load-balancing, tasks, many-core, pair potential.

1 Introduction

In order to reach exascale, current trends in super-computing are on low-energy consumption systems [1], with systems containing an increasing number of energy-aware processors and accelerators [2] [3]. These processors and accelerators offer more computing cores with reduced frequencies, making task optimization very demanding. A common way to extract parallelism from applications is to distribute the main computing flow into a large number of tasks [4]. Numerous runtimes [5] [6] [7] actually work this way. In addition, since its third version, OpenMP offers task support in it specification [8].

Accurate task scheduling must provide numerous tasks for one thread, leading to an important scheduler overhead when hundred of cores are considered, mainly due to finding the best queue and inserting the new task. Algorithm to find the best queue is critical (a bad task distribution drives to poor performances) and depends on tasks properties, e.g., average task duration, data amount, etc. Usually, those kinds of difficulties are solved by introducing work-stealing strategies, but finding a victim among thousands of threads is very expensive. Advanced

schedulers take care about data dependencies and data locality [9]. Tasks with strong data affinity should be scheduled to the same computing unit to prevent from data migrations and improve cache usage. NUMA-aware allocations. One other important aspect of tasks scheduling is the ability to take into account the possible evolution of task load during simulations.

Aiming to propose an efficient task scheduler with NUMA-aware allocations, we use a partition of the simulation domain into boxes of fixed size. Each box is associated to one elementary task and contains a few numbers of elementary calculation element, typically 10 to 100 atoms, finite elements, or finite difference cells. We then gather the boxes around a virtual center by using a Voronoï tessellation, and associated each Voronoï zone to a thread. In doing so, we ensure that threads are always dealing with a compact set of boxes, which maximize caches usage. Since the CPU cost of a task may vary due to internal evolution of the elementary calculation element, the amount of calculation of a thread could strongly vary during a simulation. Noting that the density of Voronoï centers is related to the number of elementary tasks in the Voronoï zones (the higher the density the lower the tasks number), we chose to move the Voronoï centers to adapt the CPU charge of the threads. In opposition to a "task by task" scheduling, the proposed approach induces a limited fraction of tasks to be re-scheduled during charge adaptation (typically, tasks at thread domain boundary). The method to adapt the thread charge "on the fly" uses an analogy with the dynamic of electrically charged particles.

We will first define the "virtual core" as the center of the Voronoï tessellation method, then recall the pair potential theory, and put forward the advantage of this method in tasks scheduling. We then discuss our choice of pair forces and their relevance for load balancing. A large set of test-cases, which present different charge variations (smooth/aggressive), are then proposed to demonstrate the advantages of a dynamic load balancing based on pair potentials. We will then conclude by the evaluation of our scheduler in a real parallel application.

2 Tasks Scheduling with Pair Potentials

We use a 2D grid (see fig. 1) in which every cell represents a task. Each task has a computing load of its own, which can evolve over time. We gather tasks around a virtual core (termed in the following a *vCores*, a virtual representation of the physical computing units) by using a Voronoï tessellation[10]. By this way, we maximise per-core data locality. In a shared-memory environment, this guarantees NUMA-aware allocations and better caches usage. In a distributed-memory environment, this reduces inter-node data displacements. The load of a *vCore* is the sum of the computing load of each tasks in its VoronoÏ cell. Thus, real-time tasks cost variations have a direct influence on the *vCores* load.

We then associate to each *vCores* a force and make them *virtually* move over the task domain in response to this force. By moving, a core will change its computing load, which gives the opportunity to re-equilibrates the computing load between *vCore*.

Figure (1) gives a representation of the elementary tasks (or cells) the *vCores* and the associated Voronoï cells, as colored domains.

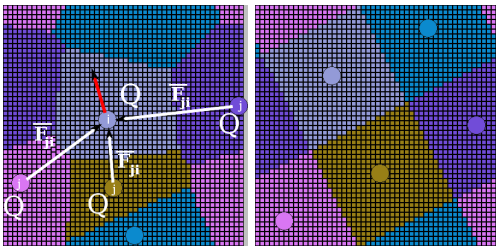

Fig. 1. By introducing pair potentials in task scheduling, a core can move over the tasks domain (red arrow). Q is the local computing load of the core and F_{ij} the applied forces on i. This local computing load modulate the intensity of attractive/repulsive forces between cores. In this case, all tasks have the same computing cost.

In the next section we define a pair potential between *vCores* that would lead to a good load balancing, for any variations of the underlying elementary tasks load.

3 Evolution of a set charged particles

Let us consider the Coulomb force between of two particles i, j separated by a vector $\mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$ with charge Q_i, Q_j:

$$\mathbf{F}_{ij} = Q_i Q_j \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}. \tag{1}$$

Different charge signs lead to attractive forces whereas charges with same sign produce repulsive ones. Pairs interactions of this N-Body system are calculated by exploiting the symmetry of interactions, i.e., $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$. In order to get the relaxed state only, we minimize the potential by using a steepest descent algorithm: $\mathrm{d}\mathbf{x}/\mathrm{d}t = -\alpha \mathbf{F}_{ij}$ (with α a positive scalar), and by lumping α and time increment into a simple scalar k:

$$\mathbf{x}(t) = \mathbf{x}(t-1) - k \sum_j \mathbf{F}_{ij} \tag{2}$$

We rescale k so that the distance $\mathbf{x}(t) - \mathbf{x}(t-1)$ is a fraction of the verlet box dimension (a task), which ensure convergence to stable or metastable states.

Going on this analogy between charged particles and computing loads, we set the charge of a *vCore* to be the sum of the computing load of each tasks in the Voronoï cell q_k:

$$Q_i = \sum_{k \in Voronoï(i)} q_k. \tag{3}$$

Preliminary results using this force are discussed in the following section. Enhanced forces expressions are then proposed to improve the load balance, and discussed.

4 Potential test cases

Despite its apparent simplicity, minimizing a set of particles interacting by an electric potential could lead to complex behaviors. For example, when homogeneous repulsive charges are considered, the minimization leads to a cubic close-packed lattice with a minimal number of neighboring cells (i.e. 12). Dealing with non-constant loads requires to slightly modify the pair potential, as proposed below, but also to test it on standardized tests cases.

We have developed a *C++* simulator which helps us to select an efficient potential. This simulator generates a grid of tasks (of different charges), and randomly inserts a bunch of *vCores* (see fig. 2). Thanks to this simulator, we have a real-time feedback on the actual load balancing and Voronoï cells configuration. Various charge evolutions are supported by our simulator. We can generate a whole new map that leads to strong tasks charge variations, or we can translate the map (smooth tasks charge transitions). The map is based on a Perlin noise [11] generated with the LibNoise [12] library. A stable configuration is reached when the *vCores* are stable (i.e. velocity is null).

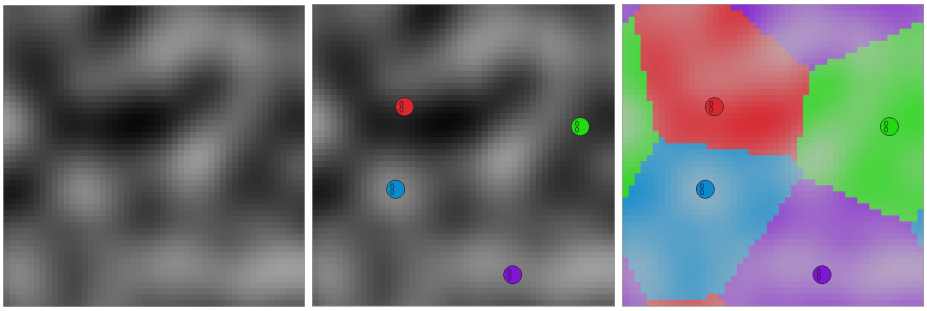

Fig. 2. Simulator used to find an efficient potential for tasks scheduling. From left to right: tasks grid with different loads, *vCores* are positioned on the grid, associated Voronoï cell.

4.1 Three Potentials

Our original idea was to use a slightly modified Coulomb potential so that overloaded *vCores* allows their neighboring *vCores* to get closer, whereas underloaded *vCores* are strongly repulsive. Two possible choices are presented. Decreasing repulsive force for increasing load Q can be obtained by considering the force:

$$\mathbf{F}_{ij} = \lambda \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3} \text{ with } \lambda = \frac{1}{Q_i} + \frac{1}{Q_j} \,. \tag{4}$$

This modified potential is repulsive-only. Preliminary tests show that minimization leads to cubic closed-packed lattice for homogeneous task load, but fails to obtain a reasonably well balanced load for inhomogeneous task repartition. With the objective to define a potential that produce null forces when *vCores* are *optimally* charged, we note that ideal load partitions are obtained when all the charges Q_i are equal to the mean charge m

$$m = \frac{1}{N} \sum_{i=1,N} Q_i \text{ with } N, \text{ the number of VCores.} \tag{5}$$

We then propose to use a potential that leads to null forces when $Q_i = m$:

$$\mathbf{F}_{ij} = \lambda \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3} \text{ with } \lambda = 1 - \frac{Q_i + Q_j}{2m}. \tag{6}$$

In this case, when a pair of *vCores* is globally under-loaded (the λ term is positive), the two *vCores* repulse each other. By this way, its Voronoï surface and local load will grow. The reverse behavior occurs when the *vCore* is over-loaded (the λ term is negative): the Voronoï surface and the local load decrease thanks to the attractive forces. Even if we have noted good load partitioning (6% to the optimal load distribution), our preliminary simulations show the formation of dipoles (two very close *vCores*). This leads to bad Voronoï partitioning (see fig. 3): some *vCores* are no longer in the center of their Voronoi cell, but close to one of the frontier.

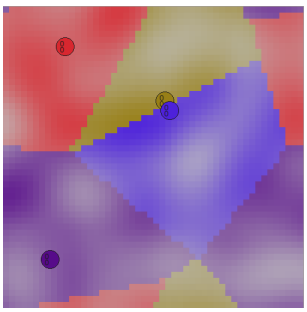

Fig. 3. Dipole formation: in this case (potential (6)) produces a bad Voronoï cell splitting. Some *vCores* are too close to each other. The short repulsive term introduced in (7) solves this issue.

To tackle to this problem, we have added a short-distance repulsion to our potential (7). This term ensures that two *vCores* can not be too close to each other.

$$\mathbf{F}_{ij} = \lambda \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3} + \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^5} \text{ with } \lambda = 1 - \frac{Q_i + Q_j}{2m}. \tag{7}$$

5 Experiments

In order to evaluate our task scheduling method, we have developed four kinds of test cases (see fig. 4). The simulated domain is a 50×50 grid of tasks, and we arbitrary place 10 *vCores* on the domain (the random position is the same for each test case). A stable configuration is accepted when the variance to the optimal task distribution is below 2. The tests runs on an Intel® Xeon E5-2650.

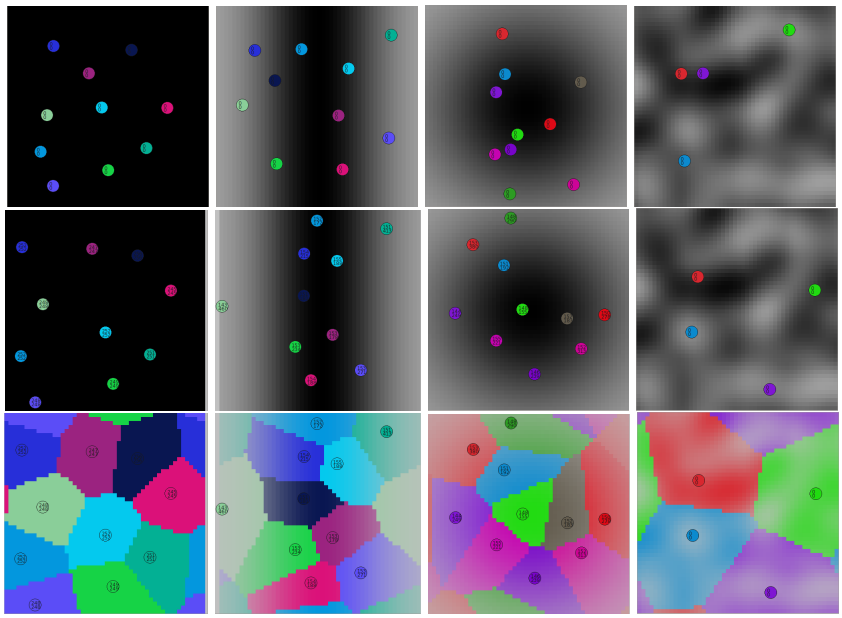

Fig. 4. From left to right: (1) the load is uniform over the domain, (2) the load is distributed over a line, (3) the load is concentrated in a disc, (4) the load is randomly distributed. From top to bottom: the first line presents the load distribution, the second line shows the final *vCores* configuration, and the last line represents the domain of each *vCores*.

5.1 Experimental Results

Static scheduling Here, we evaluate the number of computing steps needed to reach a stable *vCore* configuration. Figure 5 shows the convergence curves for our test-cases. The two curves represent the distance to the optimal load per *vCore*.

The upper curves (in blue) show the convergence of the most over-loaded *vCore*, and lower curves (in yellow), the convergence of the least under-charged *vCore*. We can observe that tests-cases (2) and (3) are complex to schedule. With the loaded-line, we never reach a good tasks scheduling: we are nearly 25% to the optimal. A solution that may solve this issue is presented in 5.2. In case of the loaded-disc, we reach a good tasks distribution, but with an important number of steps. Cases (1) and (4) reach a good tasks distribution in a reasonable number of steps.

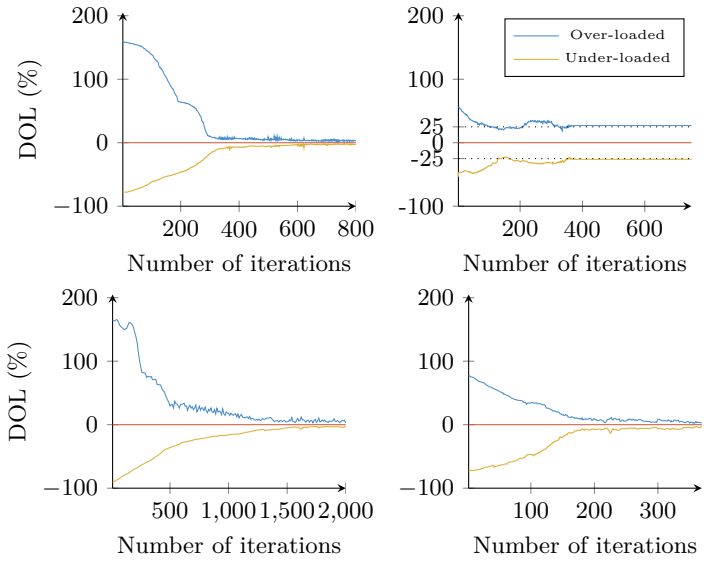

Fig. 5. Convergence of different test-cases. The red horizontal curve represents the optimal per-core load of the domain. The blue and yellow curves show the distance to optimal load (DOL) of *vCores*. This expresses the tasks distribution efficiency (tasks distribution is better when DOL is close to 0). The blue one is for the most overloaded *vCores* and the yellow one for less underloaded *vCores*. **(1)**: the load is uniform over the domain. **(2)** the load is distributed in a line, **(3)**: the load is concentrated in a disc. **(4)**: the load is randomly distributed.

Dynamic scheduling: For dynamic scheduling, we use two kinds of charge variations. The first one is the LibNoise [12] ability to change the frequency of the generated noise. By this way, we translate the load map over the task domain. The second type of load variation is done by generating a completely new map. We call rough load variation the generation of a new map, and smooth load variation a simple change in frequency of the actual noise. Table 1 summarize the efficiency of our tasks scheduling method. This shows us that reaching a

tasks distribution in case of smooth task load variation is nearly 90 times faster than in case of rough load transition.

Table 1. Average number of steps for rough and smooth charge variation over 1000 tests.

Transition type	Number of steps			Time (ms)		
	Average	Min	Max	Average	Min	Max
Rough	432.700	27	5115	32.674	3.888	755.744
Smooth	5.238	2	312	0.151	0.144	70.847

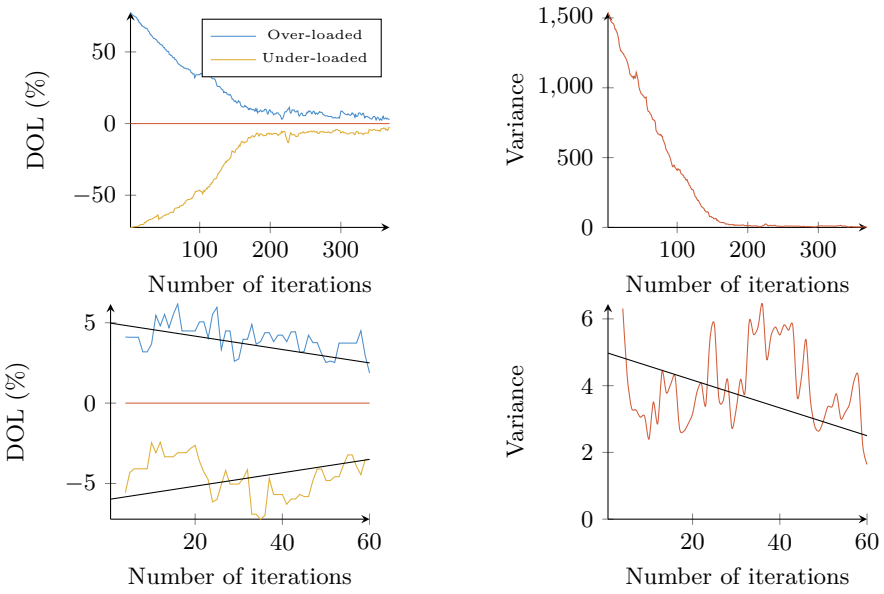

Fig. 6. Left: number of steps needed to converge after a rough (top) or a smooth (bottom) load variation of tasks for the most/least loaded *vCores*. Right: variance variations of the global system (convergence criteria).

Large number of cores: Here, we evaluate the ability to find a stable *vCores* configuration for a large number of *vCores*. We use different configurations (i.e. number of tasks, number of *vCores*) to stress our simulations. Tables 2 and 3 compare the number of steps and the time needed to distribute tasks over *vCores*. As expected, the number of *vCores* and tasks impact the number of steps needed to reach an equilibrium. In every cases, reaching a new task distribution after a smooth load variation of tasks is almost-instantaneous.

Table 2. Dynamic load distribution for a large number of *vCores*

Number of *vCores*	Domain dimensions	Transition type	Number of steps			Time (ms)		
			Average	Min	Max	Average	Min	Max
13	50x50	Rough	578.810	96	1939	138.871	29.005	599.621
	50x50	Smooth	16.229	2	1188	0.624	0.304	374.893
	100x100	Rough	997.230	238	2544	113.442	268.578	2271.620
	100x100	Smooth	26.983	2	1331	10.587	1.306	339.403
	150x150	Rough	1146.040	237	1977	279.070	582.276	4582.276
	150x150	Smooth	101.386	2	1578	68.887	2.866	1073.940

With an higher number of cores (table 3), the number of compute steps needed is reasonably proportional to the number of *vCores*. Nevertheless, the associated time explodes, due to our Voronoï tessellation algorithm. Our implementation has a complexity in $\mathcal{O}(n \times m)$, with n the number of *vCores*, and m the number of tasks in the domain. We are currently looking for graph partitioning optimisation[13] in aim of reducing this computing cost.

Table 3. Dynamic load distribution for a large number of *vCores*

Number of *vCores*	Domain dimensions	Transition type	Number of steps			Time (ms)		
			Average	Min	Max	Average	Min	Max
72	50x50	Rough	763.321	5	7784	612.415	130.010	1097.750
	50x50	Smooth	62.366	2	4942	2.174	2.068	990.199
	100x100	Rough	2310.150	477	4915	1447.130	301.027	3245.290
	100x100	Smooth	319.010	2	1988	161.877	6.289	125.502
	150x150	Rough	3152.600	803	4968	4138.220	1086.650	6760.730
	150x150	Smooth	414.260	4	1990	173.868	40.390	269.675

5.2 Complex cases

We have seen in the second case in fig. 5 that reaching a good task distribution can be difficult. The problem is that some *vCores* are heavily over-loaded while others are strongly under-loaded and are caught by surrounding *vCores*. We are currently working on a complementary potential that produces only attractive forces between tasks and *vCores*. By this way, *vCores* will be attracted by the most costly tasks. Preliminary results show promising configurations (see fig.7). Nevertheless, with this new interaction, computing time increases dramatically. We need to compute interaction between *vCores* and tasks. Initial complexity of the algorithm is in $\mathcal{O}(N^2)$, but with this potential, it increases in $\mathcal{O}(mN^2)$, with m, the number of tasks.

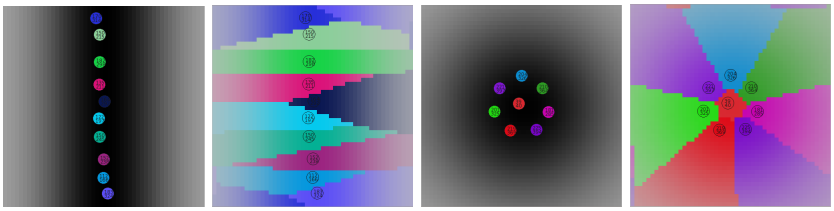

Fig. 7. With the attraction of tasks, the final distribution is better.

6 Case Study: Coddex

In order to experiment our scheduler in a real-world application, we have extended the StarPU [5] runtime, which is used by Coddex, a CEA software. Coddex is a Finite Element code dedicated to the modeling of plasticity and phase transition on solid materials. This software is based on the MPI library for inter-node communications and on StarPU for a threads/tasks parallelism inside a computing node. The experimental platform used for our experiments is a double-sockets node (Intel Xeon E5-2650). By using a multi-sockets node, we want to evaluate the ability for our scheduler to minimize the tasks data displacements over the different NUMA nodes. Our scheduler is compiled as a separate library (**P**otential **B**ased **S**cheduler, PBS) and can then be used by several applications. In order to integrate our scheduler inside StarPU, we've add a "meta-scheduler" that calls the PBS library to retrieves the tasks distribution over threads.

In the following, we compare two schedulers : eager [14], the default StarPU scheduler (a simple FIFO list of tasks) and PBS, our scheduler. We measure the average number of data cache misses [4] of each task for each simulation time step. Figure 8 presents the percent of L2 data cache misses for a typical Coddex execution (50K points, 2D domain), and the table 4 shows the cache misses rate reduction. Each square of the figure represents a task (Verlet box), and we can see that the task distributions (top of the figure) over threads (one color per thread). We can notice that the PBS scheduler provides a better memory accesses between our tasks; this is mainly due to a better cache usage, and a better NUMA accesses.

Table 4. Comparison of cache misses (L2_DCM)

		Values (%)		
		Average	Max	Min
Scheduler	Eager	5.0184	16.9964	0.0
	PBS	0.6830	3.6833	0.0
Gain (%)		86.4	78.3	0.0

[4] We use the PAPI[15] library

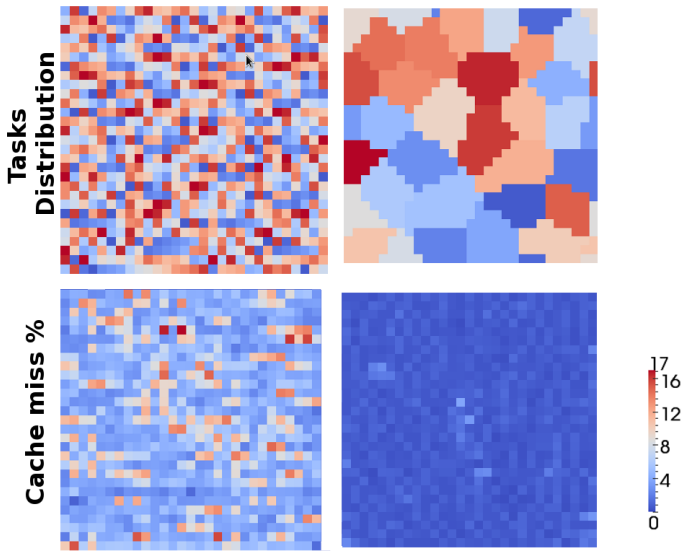

Fig. 8. Average number of L2 data cache misses for a typical Coddex execution (1 MPI node, 32 threads). The left side figures show the results with the default StarPU scheduler (eager), and the right side figures, the results with our scheduler (PBS). On the top of the figure, each color represent a thread (one per thread), while on the bottom on the figure, a color represent a cache miss rate.

7 Conclusions and Future Work

The Pair-potential approach for task scheduling over a large number of cores produces efficient task distribution, especially in case of dynamic load. Our preliminary experiments show a rapid convergence of task distribution in a close to optimal per-core load. Nevertheless, this distribution can be sometime difficult to reach: cases like the disc of load, have a complex load distribution to be balanced over *vCores*. In other more realistic cases (with a diffused load), a stable configuration of *vCores* is easy and relatively fast to compute. Best results are obtained for smooth load variations. In this case, task scheduling is nearly instantaneous. This is particularly interesting in simulations where the load is *moving* through the simulated domain (eg. shock waves). Our experimentation on a real simulation application shows an real improvement of the cache misses rates in comparison with the default StarPU scheduler.

Our next works will focus on improving the number of steps and the time needed to reach a stable configuration, and thus, by adjusting our potential and by implementing an efficient Voronoï algorithm. Some work needs to be done on the removal of some centralized aspects of the current algorithm: our actual potential needs to know the total charge of the simulated domain. This implies communications/synchronizations steps.

8 Acknowledgements

We wish to thank Julien Roussel for his participation to the forces equation 4, which have strongly accelerated our work. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS (see `https://plafrim.bordeaux.inria.fr/`).

References

1. Hemmert, S.: Green hpc: From nice to necessity. Computing in Science & Engineering **12**(6) (2010) 0008–10
2. Showerman, M., Enos, J., Pant, A., Kindratenko, V., Steffen, C., Pennington, R., Hwu, W.m.: Qp: a heterogeneous multi-accelerator cluster. In: Proc. 10th LCI International Conference on High-Performance Clustered Computing. (2009)
3. Kindratenko, V.: Novel computing architectures. Computing in Science and Engineering **11**(3) (2009) 54–57
4. Turek, J., Schwiegelshohn, U., Wolf, J.L., Yu, P.S.: Scheduling parallel tasks to minimize average response time. In: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (1994) 112–121
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2) (2011) 187–198
6. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in intel threading building blocks. Intel Technology Journal **11**(4) (2007)
7. Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science: Cilk 5.4.6 Reference Manual. (November 2001)
8. OpenMP-Committee: Openmp application program interface 3.0. Technical report
9. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In: 16th International Conference on Parallel and Distributed Systems, Shangai, Chine (December 2010)
10. Aurenhammer, F., Klein, R.: Voronoi diagrams. Handbook of Computational Geometry (2000) 201–290
11. Perlin, K.: Perlin noise is a computer-generated visual effect developed by ken perlin, who won an academy award for technical achievement for inventing it.
12. Bevins, J.: Libnoise: a portable, open-source, coherent noise-generating library for c++.
13. Chevalier, C., Pellegrini, F.: Pt-scotch: A tool for efficient parallel graph ordering. Parallel Computing **34**(6?8) (2008) 318 – 331 Parallel Matrix Algorithms and Applications.
14. Sarmenta, L.F.G., Hirano, S.: Bayanihan: Building and studying web-based volunteer computing systems using java. **15** (1999) 675–686
15. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3) (August 2000) 189–204